APPENDIX I:

NLP++ INTEGRATION WITH A KNOWLEDGE BASE

The knowledge base may consist of a hierarchy of concepts (CON). Each concept may have a set of attributes (ATTR). Each attribute may have a name and a value (VAL). The value may be a string (STR), number (NUM), boolean (BOOL), pointer to another concept, or some other type. Each ATTR may have multiple values, and each value may have a distinct type.

Each concept may also have an associated phrase (PHR) of nodes. A node may be similar to a concept in most respects, except, for example, a node may never be placed directly in the knowledge base hierarchy. Rather, nodes may serve as proxies or references to concepts that are in the hierarchy. Phrases may be used to implement idioms, patterns, samples, rules, unordered sets of concepts or any other information.

The objects discussed above, CON, ATTR, VAL and PHR, may be assigned as types of values of NLP++ variables. These functions may treat nodes as concepts.

The functions in **Table XII** enable accessing and manipulating the objects of the knowledge base. These functions illustrate integrating the NLP++ language with a knowledge base to build a text analyzer.

Table XII. Exemplary Functions Associated with Accessing and Manipulating Objects of the Knowledge Base.

FUNCTION	RETURN TYPE	DESCRIPTION
FETCH OBJECTS		
findroot()	CON	Return the root concept of the knowledge base (named "concept").
findconcept(*parent, name***)**	CON	Find the named concept under the specified parent concept.
findconcept(*parent, num***)**	CON	Find the *num*-th concept under the specified parent concept.
findattr(*con, name***)**	ATTR	Find the named attribute under the specified concept.
findattrs(*con***)**	ATTR	Fetch the specified concept's list of attributes.
attrname(*attr***)**	STR	Fetch the specified attribute's name.
attrvals(*attr***)**	VAL	Fetch the specified attribute's values.
findvals(*con, name***)**	VAL	Fetch the values in the specified concept's named attribute.
numval(*con, name***)**	NUM	Fetch the numeric value of the named attribute.
strval(*con, name***)**	STR	Fetch the string value of the named attribute.
conval(*con, attr_str***)**	CON	Fetch the concept-value of the specified concept's attribute. (The concept-value must be first value).

attrwithval(*con, attr_s, val_s***)**	BOOL	Determine whether the specified attribute has the specified value.
inheritval(*con, name, hier***)**	STR	Find the string value of an attribute having the specified name, searching from the concept *con* up to the concept *hier*. (A predetermined special value for *hier* may specify the root of the knowledge base (KB).)
conceptname(*con***)**	STR	Fetch the name of the specified concept.
conceptpath(*con***)**	STR	Return the entire path of the specified concept as a string.
pathconcept(*str***)**	CON	Fetch the concept specified by the path *str*.
wordpath(*str***)**	STR	Fetch the entire path of word-concept for the specified string.
findwordpath(*str***)**	STR	Find entire path of word-concept for the specified string. (If not present, don't add the word.)
wordindex(*str***)**	CON	Get index concept that str would be added under.
findhierconcept(*name, hier***)**	CON	Find the named concept in the sub-hierarchy of the specified concept. (A predetermined special value for *hier* may specify the root of the knowledge base (KB).)

dictfindword(*str*)	CON	Find the named concept in the dictionary hierarchy of the KB.
attrexists(*hier*, *attr_s*, *val_s*)	BOOL	Determine whether the specified attribute-value pair exists in the specified sub-hierarchy.
attrchange(*hier*, *attr_s*, *val_s*, *new_s*)	BOOL	Change all the attributes in the specified hierarchy having the specified name and value to have the specified new string value.
down(*con*)	CON	Fetch the first child of the specified concept.
up(*con*)	CON	Fetch the parent of the specified concept.
prev(*con*)	CON	Fetch the left or previous sibling of the specified concept.
next(*con*)	CON	Fetch the right or next sibling of the specified concept.
nextattr(*attr*)	ATTR	Fetch the next attribute in a list of attributes.
nextval(*val*)	VAL	Fetch the next value in a list of values.
getsval(*val*)	STR	Convert the specified value to a string, if possible.
getstrval(*val*)	STR	Get string from string-valued *val*.
getnumval(*val*)	NUM	Get number from numeric *val*.
getconval(*val*)	CON	Get concept from concept-valued *val*.
MAKE OBJECTS		

makeconcept(*parent, name, num***)** **makeconcept(***parent, name***)**	CON	Place the named concept under the specified parent concept. If *num* is non-zero, the named concept becomes the *num*-th child of the parent. If *num* is zero or absent, places the named concept at end of the list of children.
addattr(*con, attr_s***)**	ATTR	To the specified concept, add the specified attribute with no value.
addsval(*con, name, num***)**	-	Add the specified numeric value as a string to the specified attribute.
addstrval(*con, name, str***)**	-	Add the specified string value to specified attribute.
addnumval(*con, name, num***)**	-	Add the specified numeric value to the specified attribute.
addconval(*con, attr_str, value_con***)**	-	Add the specified value (*value_con)* as the value of specified attribute.
getconcept(*parent, name***)**	CON	Find the named concept under the specified parent. If the named concept does not exist, this functions operates as **makeconcept(***parent, name***)**.
addword(*str***)**	CON	Add the specified word to the dictionary within the KB, if the word is not already present. Also, fetch the dictionary concept for the word.
MOVE & REMOVE		

rmconcept(*con***)**	BOOL	Remove (the entire sub-hierarchy of) the specified concept from the KB.
rmchild(*parent*, *name***)**	BOOL	Remove the named child of the specified parent concept.
rmchild(*parent*, *num***)**	BOOL	Remove the *num*-th child of specified parent concept.
rmvals(*con*, *name***)**	BOOL	Remove the values of the specified attribute.
rmval(*attr*, *val***)**	BOOL	Remove the specified value from the specified attribute.
rmattrval(*con*, *attr_s*, *val_s***)**	BOOL	Remove the specified string value from the specified attribute.
rmattr(*con*, *name***)**	BOOL	Remove the named attribute (and its values) from the specified concept.
rmchildren(*con***)**	BOOL	Remove the children (and phrase) from the specified concept.
rmword(*word_str***)**	BOOL	Remove the specified word from the KB dictionary.
prunephrases(*hier***)**	BOOL	Remove all phrases from the specified sub-hierarchy.
replaceval(*con*, *name*, *str***)**	-	Replace all of the values of the specified attribute with the specified string.
replaceval(*con*, *name*, *num***)**	-	Replace all of the values of the specified attribute with the specified number.
replaceval(*con*, *attr_str*, value_con**)**	-	Replace the specified attribute's value with the specified value (*value_con*).

renameconcept(*con, name***)**	-	Rename the specified concept to the specified name.
renamechild(*con, num, name***)**	-	Rename the *num*-th child of the specified concept to the specified name.
renameattr(*con, name, new***)**	-	Rename the specified attribute to the specified name (*new*).
movecleft(*con***)**	-	Move the specified concept one to the left in its list.
movecright(*con***)**	-	Move the specified concept one to the right in its list.
KNOWLEDGE-BASE PHRASES & NODES		
findphrase(*con***)**	PHR	Fetch the specified concept's phrase.
sortphrase(*con***)**	-	Alphabetically sort the specified concept's phrase nodes.
phraselength(*con***)**	NUM	Get the number of nodes in the phrase of the specified concept.
nodeconcept(*node***)**	CON	Get the owning concept of the specified node concept.
findnode(*phrase, name***)**	CON	Find the first node with the specified name in the specified phrase.
findnode(*phrase, num***)**	CON	Find the *num*-th node in the specified phrase.
listnode(*node***)**	CON	Get the first node in the specified node's list.
firstnode(*phrase***)**	CON	Get the first node in the specified phrase.
lastnode(*phrase***)**	CON	Get the last node in the specified phrase.

makephrase(*con*, *name***)**	PHR	Create a phrase in the specified concept by making the named node.
addcnode(*con*, *name***)**	CON	Make the named node at the end of the specified concept's phrase.
addnode(*phrase*, *name*, *num***)**	CON	Make the named node the *num*-th in the specified phrase.
rmnode(*con***)**	-	Remove the specified node from its phrase.
rmphrase(*phrase***)**	-	Remove the specified phrase from its concept.
rmcphrase(*con***)**	-	Remove the phrase of the specified concept.
renamenode(*phrase*, *name*, new**)**	-	Rename the specified phrase's named node to *new*.
renamenode(*phrase*, *num*, new**)**	-	Rename the specified phrase's *num*-th node to new.

APPENDIX II:
RULE-FILE ANALYZER

This appendix defines a text analyzer that a shell may use to read pass files of a user-built analyzer. This appendix contains files in the order in which they are read, the same order in which the rule-file analyzer may be executed.

The first file, analyzer.seq, defines the sequence of passes in the analyzer. Each line in that file consists of the name of an algorithm for the pass — for example, "pat," the main pattern-based algorithm. Each line also contains data associated with the pass. For example, "retok" refers to the retok pass file associated with the third pass.

The rule-file analyzer uses special functions for constructing the internal machinery of a text analyzer. Example functions are rfaname(), rfaop(), rfastr(), rfarulemark(), rfanonlit(), rfanum(), rfanodes(), rfaarg(), rfalist(), rfarange(), rfaexpr(), rfaunary(), rfapostunary(), rfaargtolist(), rfapair(), rfalittopair(), rfapairs(), rfaelement(), rfanonlitwlt(), rfalitelt(), rfasugg(), rfaelt(), rfarule(), rfarulelts(), rfarules(), rfaactions(), rfapres(), reaselect(), rfaregion(), rfaregions(), rfarecurse(), rfarecurses() and rfarulesfile(). They may build an optimized internal semantic representation orthogonal to the semantic variables that a user may add to parse-tree nodes.

The rule-file analyzer, which analyzes NLP++, is itself defined using a subset of the full NLP++ language:

```
###############################################
            # FILE:     ANALYZER.SEQ
###############################################
#
tokenize            #
line        #
pat   retok #
pat   bigtok      #
pat   x_white     #
pat   nlppp #
pat   un_mark     #
pat   list  #
pat   list1 #
pat   gram1 #
rec   gram2 #
pat   preaction   #
pat   gram4 #
rec   gram5 #
pat   action      #
pat   pair  #
pat   pairs #
pat   element     #
pat   rule  #
pat   rules #
pat   code  #
pat   pres  #
pat   checks      #
pat   posts #
pat   tmp   #
pat   tmp1  #
pat   select      #
pat   region      #
pat   regions     #
pat   recurse     #
pat   recurses    #
pat   rulesfile   #
nintern     nil   #
gen   nil   #
hash  nil   #
genhash     nil   #
```



```
################################################
              # FILE:     RETOK.PAT
################################################

# Since RFB rules are hashed, don't need sentinel.
#@POST
#     noop()
#@RULES
#_xNIL <- _xWILD [fail=(\\)] @@

@POST
      rfaname(2)
      single()
@RULES
_cLF [base layer=(_LIT )] <- \\ n [ren=\n] @@
_cCR [base layer=(_LIT )] <- \\ r [ren=\r] @@
_cHT [base layer=(_LIT )] <- \\ t [ren=\t] @@

_cLANGLE [base layer=(_LIT )] <- \\ \< @@
_cPOUND [base layer=(_LIT )] <- \\ \# @@
_cDQUOTE [base layer=(_LIT )] <- \\ \" @@
_cATSIGN [base layer=(_LIT )] <- \\ \@ @@
_cLPAR [base layer=(_LIT )] <- \\ \( @@
_cRPAR [base layer=(_LIT )] <- \\ \) @@
_cCOMMA [base layer=(_LIT )] <- \\ \, @@
_cSEMICOLON [base layer=(_LIT )] <- \\ \; @@
_cEQUAL [base layer=(_LIT )] <- \\ \= @@
_cLBRACKET [base layer=(_LIT )] <- \\ \[ @@
_cRBRACKET [base layer=(_LIT )] <- \\ \] @@
_cUNDERSCORE [base layer=(_LIT )] <- \\ \_ @@
_cDASH [base layer=(_LIT )] <- \\ \- @@
_cSPACE [base layer=(_LIT )] <- \\ \  @@
_cRANGLE [base layer=(_LIT )] <- \\ \> @@

_cBEL [base layer=(_LIT )] <- \\ a [ren=\a] @@
_cBS [base layer=(_LIT )] <- \\ b [ren=\b] @@
_cFF [base layer=(_LIT )] <- \\ f [ren=\f] @@
_cVT [base layer=(_LIT )] <- \\ v [ren=\v] @@
_cSQUOTE [base layer=(_LIT )] <- \\ \' @@
_cQMARK [base layer=(_LIT )] <- \\ \? @@
_cBANG [base layer=(_LIT )] <- \\ \! @@
_cDOLLAR [base layer=(_LIT )] <- \\ \$ @@
_cPERCENT [base layer=(_LIT )] <- \\ \% @@
_cAMPERSAND [base layer=(_LIT )] <- \\ \& @@
_cASTERISK [base layer=(_LIT )] <- \\ \* @@
_cPLUS [base layer=(_LIT )] <- \\ \+ @@
```



```
_cPERIOD [base layer=(_LIT )] <- \\ \. @@
_cSLASH [base layer=(_LIT )] <- \\ \/ @@
_cCOLON [base layer=(_LIT )] <- \\ \: @@
_cCARET [base layer=(_LIT )] <- \\ \^ @@
_cBACKQUOTE [base layer=(_LIT )] <- \\ \` @@
_cLBRACE [base layer=(_LIT )] <- \\ \{ @@
_cRBRACE [base layer=(_LIT )] <- \\ \} @@
_cVBAR [base layer=(_LIT )] <- \\ \| @@
_cTILDE [base layer=(_LIT )] <- \\ \~ @@
_cBSLASH [base layer=(_LIT )] <- \\ \\ @@

@POST
      excise(1,1)
@RULES
_xNIL <- \r \n @@
```



```
###############################################
# FILE:     BIGTOK.PAT
###############################################

@POST
        excise(1, 3)
@RULES
_xNIL <- \# _xWILD \n @@

@POST
        excise(1, 2)
@RULES
_xNIL <- \# _xWILD _xEOF @@

@POST
        rfastr(2)
        single()
@RULES
_STR [base] <- \" _xWILD \" @@

#@POST
#       excise(1, 1)
#@RULES
#_xNIL <- \, [plus] @@

# EXPRESSION GRAMMAR.
@POST
        rfaop(1,2)
        single()
@RULES
_opAND <- \& \& @@
_opOR  <- \| \| @@
_opINC <- \+ \+ @@
_opDEC <- \- \- @@
_opEQ  <- \= \= @@
_opNEQ <- \! \= @@
_opGE  <- \> \= @@
_opLE  <- \< \= @@
_opCONF <- \% \% @@
_opOUT <- \< \< @@

@RULES
_ENDRULE [base] <- \@ \@ _xWHITE @@

#@POST
#       noop()
```



```
#@RULES
#_xNIL <- _xWILD [min=1 max=1 fail=(\@)] @@

@RULES
_ENDRULE [base] <- \@ \@ _xEOF @@
_eoPOST [base layer=(_endMark)] <- \@ \@ POST [t] @@
_eoCHECK [base layer=(_endMark)] <- \@ \@ CHECK [t] @@
_eoPRE [base layer=(_endMark)] <- \@ \@ PRE [t] @@
_eoRULES [base layer=(_endMark)] <- \@ \@ RULES [t] @@
_eoRECURSE [base layer=(_endMark)] <- \@ \@ RECURSE [t] @@
_eoSELECT [base layer=(_endMark)] <- \@ \@ SELECT [t] @@
_eoNODES [base layer=(_endMark)] <- \@ \@ NODES [t] @@
_eoMULTI [base layer=(_endMark)] <- \@ \@ MULTI [t] @@
_eoPATH [base layer=(_endMark)] <- \@ \@ PATH [t] @@
_eoCODE [base layer=(_endMark)] <- \@ \@ CODE [t] @@
_soPOST [base layer=(_startMark)] <- \@ POST [t] @@
_soCHECK [base layer=(_startMark)] <- \@ CHECK [t] @@
_soPRE [base layer=(_startMark)] <- \@ PRE [t] @@
_soNODES [base layer=(_startMark)] <- \@ NODES [t] @@
_soMULTI [base layer=(_startMark)] <- \@ MULTI [t] @@
_soPATH [base layer=(_startMark)] <- \@ PATH [t] @@
_soCODE [base layer=(_startMark)] <- \@ CODE [t] @@
_soSELECT [base layer=(_startMark)] <- \@ SELECT [t] @@
_soRECURSE [base layer=(_startMark)] <- \@ RECURSE [t] @@

# Separating out rule mark so it can be counted.
# If there are none, then don't need to warn about no rules in pass.
@POST
  rfarulemark()
  single()
@RULES
_soRULES [base layer=(_startMark)] <- \@ RULES [t] @@

@POST
      rfanonlit(2)
      single()
@RULES
_NONLIT [base] <- \_ _xALPHA @@

@RULES
_ARROW [base] <- \< \- @@

@POST
      rfaname(1)
      single()
@RULES
```

```
# Not setting base for these potential keywords.
_LIT <- _xWILD [s one match=(
      N X G P s
      if else while
      )] @@

_LIT [base] <- _xALPHA @@

@POST
      rfanum(1)
      single()
@RULES
_NUM [base] <- _xNUM @@
```

```
##################################################
# FILE:     X_WHITE.PAT
##################################################
@POST
      excise(1, 1)
@RULES
_xNIL <- \  @@
_xNIL <- \n @@
_xNIL <- \t @@
```



```
###############################################
# FILE: NLPPP.PAT
# SUBJ: Creating regions for parsing NLP++ syntax and others.
# NOTE: Code regions are parsed differently from the rules.
###############################################
@POST
      rfanodes(2, "nodes")
      single()
@RULES
_NODES [base] <- _soNODES [s] _NONLIT [star] _eoNODES [s opt] @@

@POST
      rfanodes(2, "path")
      single()
@RULES
_PATH [base] <- _soPATH [s] _NONLIT [star] _eoPATH [s opt] @@

@POST
      rfanodes(2, "multi")
      single()
@RULES
_MULTI [base] <- _soMULTI [s] _NONLIT [star] _eoMULTI [s opt] @@

@POST
  group(2,2, "_NLPPP")  # An NLP++ region.
  singler(1,3)
@RULES
_PRES [base unsealed] <-
      _soPRE [s]
      _xWILD [fail=(_endMark _startMark)]
      _eoPRE [s opt]
      _xWILD [opt lookahead match=(_endMark _startMark)]
      @@

@POST
  group(2,2, "_NLPPP")  # An NLP++ region.
  singler(1,3)
@RULES
_CHECKS [base unsealed] <-
      _soCHECK [s]
      _xWILD [fail=(_endMark _startMark)]
      _eoCHECK [s opt]
      _xWILD [opt lookahead match=(_endMark _startMark)]
      @@

@POST
```

```
  group(2,2, "_NLPPP")  # An NLP++ region.
  singler(1,3)
@RULES
_POSTS [base unsealed] <-
      _soPOST [s]
      _xWILD [fail=(_endMark _startMark)]
      _eoPOST [s opt]
      _xWILD [opt lookahead match=(_endMark _startMark)]
      @@

@POST
  singler(1,3)
@RULES
_RULES [base unsealed] <-
      _soRULES [s]
      _xWILD [fail=(_endMark _startMark)]
      _eoRULES [s opt]
      _xWILD [opt lookahead match=(_endMark _startMark)]
      @@

# INI REGION.  FOR RUNNING CODE BEFORE SOMETHING (like @nodes).
@POST
  group(2,2, "_NLPPP")  # An NLP++ region.
  singler(1,3)
@RULES
_INI [base unsealed] <-
      _soINI [s]
      _xWILD [fail=(_endMark _startMark)]
      _eoINI [s opt]
      _xWILD [opt lookahead match=(_endMark _startMark)]
      @@

# FIN REGION.  FOR RUNNING CODE AFTER SOMETHING (like @nodes).
@POST
  group(2,2, "_NLPPP")  # An NLP++ region.
  singler(1,3)
@RULES
_FIN [base unsealed] <-
      _soFIN [s]
      _xWILD [fail=(_endMark _startMark)]
      _eoFIN [s opt]
      _xWILD [opt lookahead match=(_endMark _startMark)]
      @@

@POST
  group(2,2, "_NLPPP")  # An NLP++ region.
```



```
  setbase(2,"true")
  singler(1,3)
@RULES
_CODE [base unsealed] <-
      _soCODE [s]
      _xWILD [fail=(_endMark _startMark)]
      _eoCODE [s opt]
      _xWILD [opt lookahead match=(_endMark _startMark)]
      @@
```



```
#################################################
# FILE: UN_MARK.PAT
#################################################
# Delete empty regions.
@NODES _CODE _CHECKS _PRES _POSTS _NODES _PATH _MULTI _RULES

@POST
  excise(1,1)
@RULES
_xNIL <- _xWILD [one match=(_startMark _endMark)] @@
```



```
#################################################
# FILE:     LIST.PAT
#################################################
@PATH _ROOT _RULES

@RECURSE listarg

@POST
      rfaarg(1)
      single()
@RULES
_ARG [base] <- _NONLIT @@
_ARG [base] <- _LIT @@
_ARG [base] <- _STR @@
_ARG [base] <- _NUM @@

@@RECURSE listarg

@POST
      rfalist(2)
      single()
@RULES
_LIST [base] <- \( _XWILD [match=(_LIT _NONLIT _STR _NUM) recurse=(listarg)]
\) @@
```



```
################################################
# FILE: LIST1.PAT
# SUBJ: For executing in NLP++ regions.
# NOTE: Code-like regions get parsed differently from rule regions.
################################################
@NODES _NLPPP

@POST
      rfarange(3, 5)
      singler(2,6)
@RULES
_PREPAIR [base] <-
      \;    # Disambiguating context.
      \< _NUM \, _NUM \> @@

_PREPAIR [base] <-
      _xSTART     # Disambiguating context.
      \< _NUM \, _NUM \> @@
```



```
###############################################
# FILE: GRAM1.PAT
# SUBJ: NLP++ sentence and expression grammar.
# NOTE: RECURSIVE PASS.
###############################################

@NODES _NLPPP

@RULES
# NLP++ KEYWORDS.
_IF [base] <- if [s] @@
_ELSE [base] <- else [s] @@
_WHILE [base] <- while [s] @@

# Binary ops.
@POST
      movesem(1)
      single()
@RULES
_OP <- _xWILD [s one match=( _opAND _opOR
      _opEQ _opNEQ _opGE _opLE
      _opCONF _opOUT    )]
      @@
```



```
##################################################
# FILE: GRAM2.PAT
# SUBJ: NLP++ code syntax.
# NOTE: RECURSIVE PASS.
##################################################
@NODES _NLPPP

# Catch the start of a function call here, so it won't be grabbed by
# expression grammar.
@POST
  fncallstart()
  single()
@RULES
_VARLIST [base] <-
      _xWILD [s one match=( S G N X P ) layer=(_VARNAME)]
      \( @@

@POST
  fncallstart()
  single()
@RULES
_FNCALLLIST [base] <- _LIT [layer=(_FNNAME)] \( @@

@POST
      movesem(2)        # Move expr semantic object up the tree.
      single()
@RULES

_EXPR <- \( _xWILD [s one match=( _EXPR _NUM _STR )] \) @@

@POST
      rfaexpr(1,2,3)
      singler(1,3)
@RULES
_EXPR <-
      _xWILD [s one match=(_EXPR _NUM _STR )]
      _xWILD [s t one match=( \* \/ \% _opCONF )]
      _xWILD [s one match=(_EXPR _NUM _STR )]
      _xWILD [s one fail=(_opINC _opDEC)]
      @@

# Handling precedence.  That's why these rules look funny.
@POST
      rfaexpr(1,2,3)
      singler(1,3)
@RULES
```

```
_EXPR <-
      _xWILD [s one match=(_EXPR _NUM _STR )]
      _xWILD [s t one match=( \+ \- )]
      _xWILD [s one match=(_EXPR _NUM _STR )]
      _xWILD [s one match=( _xANY _xEND _xEOF ) except=( \/ \* \%
                  _opCONF _opINC _opDEC )]
      @@

@POST
      rfaexpr(1,2,3)
      singler(1,3)
@RULES
_EXPR <-
      _xWILD [s one match=(_EXPR _NUM _STR )]
      _xWILD [s t one match=( \< \> _opLE _opGE _opEQ _opNEQ )]
      _xWILD [s one match=(_EXPR _NUM _STR )]
      _xWILD [s one match=( _xANY _xEND _xEOF ) except=( \/ \* \% \+ \-
            _opCONF _opINC _opDEC )]
      @@

@POST
      rfaexpr(1,2,3)
      singler(1,3)
@RULES
_EXPR <-
      _xWILD [s one match=(_EXPR _NUM _STR )]
      _xWILD [s t one match=( _opAND _opOR )]
      _xWILD [s one match=(_EXPR _NUM _STR )]
      _xWILD [s one match=( _xANY _xEND _xEOF )
            except=( \/ \* \% \+ \- \< \> _opLE _opGE _opEQ _opNEQ
                  _opCONF _opINC _opDEC )]
      @@

# Making assignment into an expr.
# LOWEST PRECEDENCE of any operator except output op (<<).

_EXPR <-
      _VAR [s]
      \= [s]
      _xWILD [s one match=( _EXPR _NUM _STR )]
      _xWILD [s one match=( _xANY _xEND _xEOF )
        except=( \/ \* \% \+ \- \< \> _opLE _opGE _opEQ _opNEQ
            _opAND _opOR
            _opCONF
            \=          # To associate right to left.
            _opINC _opDEC )]
```

```
        @@

# Output operator.
# LOWEST PRECEDENCE of any operator.

_EXPR <-
        _xWILD [s one match=(_STR _EXPR)]
        _opOUT [s]
        _xWILD [s one match=( _EXPR _NUM _STR )]
        _xWILD [s one match=( _xANY _xEND _xEOF )
          except=( \/ \* \% \+ \- \< \> _opLE _opGE _opEQ _opNEQ
                _opAND _opOR
                _opCONF
                \=
                _opINC _opDEC )]
        @@

@POST
        rfaunary(1,2)
        singler(1,2)
@RULES
# Unary operators.

# Highest precedence, apart from post operators.
_EXPR <- _xWILD [s one match=( _opINC _opDEC )]
        _VAR [s]
        _xWILD [s one match=( _xANY _xEND _xEOF) except=( _opINC _opDEC)]
        @@

_EXPR <- \! [s]
        _xWILD [s one match=( _EXPR _NUM _STR )]
        _xWILD [s one match=( _xANY _xEND _xEOF) except=( _opINC _opDEC)]
        @@

# Highest precedence operators.
@POST
        rfapostunary(1,2)
        single()
@RULES
_EXPR <-
        _VAR [s]
        _xWILD [s one match=( _opINC _opDEC )]
        @@

# Post unary ops have precedence.
@POST
```



```
        rfaunary(2,3)
        singler(2,3)
@RULES
# Only do this if you're at the start of something or there's an
# operator to the left.
_EXPR <-
        _xWILD [s one match=( _xSTART \< \> \+ \- \* \/ \% \! \=
                _opINC _opDEC _opLE _opGE _opEQ _opNE _opAND _opOR
                _opCONF
                _opOUT
                )]
        _xWILD [s t one match=( \- \+ )]
        _xWILD [s one match=( _EXPR _NUM )]
        _xWILD [s one match=( _xANY _xEND _xEOF )

                except=( _opINC _opDEC)]
        @@
```



```
# GENERALIZED FUNCTION CALL GRAMMAR.
# ---------------------------------
# LIST GRAMMAR.
# FUNCTION CALL GRAMMAR.
@POST
   addarg(1,3)
   listadd(1,3)
@RULES

_VARLIST <- _VARLIST
      \, [opt]
      _xWILD [one match=(_EXPR _NUM _STR)]
      _xWILD [one match=( \, \) )]        # lookahead.
      @@

@POST
   addarg(1,3)
   listadd(1,3)
@RULES
_xNIL <- _FNCALLLIST
      \, [opt]
      _xWILD [one match=(_EXPR _NUM _STR)]
      _xWILD [one match=( \, \) )]        # lookahead.
      @@

@POST
  varfn()
  single()
@RULES
_VAR [layer=(_EXPR)] <- _VARLIST \) @@

@POST
  movesem(1)
  single()
@RULES
_FNCALL [layer=(_EXPR)] <- _FNCALLLIST \) @@
```



```
################################################
# FILE: PREACTION.PAT
################################################
@NODES _NLPPP

@POST
   preaction()
   single()
@RULES
_ACTION [base] <- _PREPAIR _FNCALL [s] \; [s opt] @@
```



```
################################################
# FILE: GRAM4.PAT
# SUBJ: NLP++ syntax.
# NOTE: RECURSIVE PASS.
################################################
@NODES _NLPPP

@POST
      movesem(2)
      single()
@RULES

_IFPART <- _IF _xWILD [s one match=( _EXPR _NUM _STR )] @@

# Simple statements.
@POST
#     movesem(1)
      makestmt(1)
      single()
@RULES
_STMT <- _xWILD [s one match=( _EXPR _NUM _STR )] \; [s] @@

# EMPTY STATEMENT.
@RULES
_STMT <- \; [s] @@
```

```
##################################################
# FILE: GRAM5.PAT
# SUBJ: NLP++ syntax.
# NOTE: RECURSIVE PASS.
##################################################

@NODES _NLPPP

@POST
	makestmt(1)
	single()
@RULES

# NEED THE BASE, OR GRAMMAR INFINITE LOOP!
_STMTS [base] <- _xWILD [s one match=(_STMT _EXPR
	_BLOCK
	)] @@

@POST
	addstmt(1, 2)
	single()
@RULES
_STMTS [base] <- _STMTS _xWILD [s one match=(_STMT _EXPR
	_BLOCK
	)] @@

@POST
	movesem(2)
	single()
@RULES
_BLOCK <- \{ [s] _STMTS \} [s] @@

# EMPTY BLOCK.
@RULES
_BLOCK <- \{ [s] \} [s] @@

@POST
	ifstmt(1, 2)
	single()
@RULES
_IFSTMT <-
	_IFPART
	_xWILD [s one match=(_BLOCK _STMT _EXPR)]
	@@

# WHILE STATEMENT
```



```
@POST
      movesem(2)
      single()
@RULES
_WHILECOND <- _WHILE _EXPR @@

# Should make sure expr is parenthesized.
@POST
      whilestmt(1, 2)
      single()
@RULES

_STMT <- _WHILECOND     _xWILD [s one match=(_BLOCK _STMT)]
      @@

@POST
#     movesem(2)
      makestmt(2)
      single()
@RULES
_ELSEPART <-
      _ELSE
      _xWILD [s one match=(_BLOCK _STMT _EXPR)]
      @@

@POST
      ifelse(1, 2)
      single()
@RULES
_STMT <-
      _IFSTMT _ELSEPART
      @@

@POST
      movesem(1)
      singler(1, 1)
@RULES
_STMT [base] <- _IFSTMT
      _xWILD [s one match=( _xANY _xEND _xEOF ) except=( _ELSE )]
       @@
```

```
################################################
# FILE:     ACTION.PAT
################################################
@NODES _NLPPP

@POST
  setbase(1,"true")
@RULES
_xNIL <- _STMTS [s] @@

```



```
################################################
# FILE:     PAIR.PAT
################################################
@PATH _ROOT _RULES

@RECURSE listarg

@POST
      rfaarg(1)
      single()
@RULES
_ARG [base] <- _NONLIT @@
_ARG [base] <- _LIT @@
_ARG [base] <- _STR @@
_ARG [base] <- _NUM @@

@@RECURSE listarg

@RECURSE argtolist

@POST
      rfaargtolist(1)
      single()
@RULES
_LIST <- _ARG @@

@@RECURSE argtolist

@POST
      rfapair(1, 3)
      single()
@RULES
_PAIR [base] <- _LIT \= [trig] _xWILD [min=1 max=1 match=(_LIT _NONLIT
        _STR _NUM _LIST) recurse=(listarg argtolist)] @@
```



```
##################################################
# FILE:     PAIRS.PAT
##################################################
@PATH _ROOT _RULES

@RECURSE littopair

@POST
      rfalittopair(1)
      single()
@RULES
_PAIR <- _LIT @@

@@RECURSE littopair

@POST
      rfapairs(2)
      single()
@RULES
_PAIRS [base] <- \[ _xWILD [match=(_LIT _PAIR \*) recurse=(littopair)] \]
@@
```

```
################################################
# FILE:     ELEMENT.PAT
################################################
@PATH _ROOT _RULES

@POST
      rfaelement(1, 2)
      single()
@RULES
_ELEMENT [base] <- _NONLIT _PAIRS @@
_ELEMENT [base] <- _LIT _PAIRS @@
_ELEMENT [base] <- _NUM _PAIRS @@
```

```
################################################
# FILE:     RULE.PAT
################################################
@PATH _ROOT _RULES

@RECURSE rulelt

@POST
      rfanonlitelt(1)
      single()
@RULES
_ELEMENT [base] <- _NONLIT @@

@POST
      rfalitelt(1)
      single()
@RULES
_ELEMENT [base] <- _LIT @@
_ELEMENT [base] <- _NUM @@

@@RECURSE rulelt

@RECURSE sugg

@POST
      rfasugg(1)
      single()
@RULES
_SUGG <- _ELEMENT @@

@@RECURSE sugg

@RECURSE elt

@POST
      rfaelt(1)
      single()
@RULES
_ELT <- _ELEMENT @@

@@RECURSE elt

@RECURSE rulelts

@POST
      rfarulelts(1)
```



```
      single()
@RULES
_PHRASE [base] <- _ELT [plus] @@

@@RECURSE rulelts

@POST
      rfarule(1, 3)
      single()
@RULES
_RULE [base] <-
      _xWILD [one match=(_NONLIT _ELEMENT _LIT) recurse=(rulelt sugg)]
      _ARROW [trig]
      _xWILD [recurse=(rulelt elt rulelts) fail=(_ENDRULE _ARROW)]
      _ENDRULE
      @@
```



```
################################################
# FILE:     RULES.PAT
################################################
@PATH _ROOT _RULES

@POST
      rfarules(1)
      single()
@RULES
_RULES [base] <- _RULE [plus trig] @@
```

```
################################################
# FILE:     CODE.PAT
################################################
@PATH _ROOT _CODE _NLPPP

@POST
      rfaactions(1)
      single()
@RULES
_CODE [base] <- _STMTS [plus] @@
```

```
##################################################
# FILE: PRES.PAT
##################################################
@PATH _ROOT _PRES _NLPPP

@POST
      rfapres(1)
      single()
@RULES
_PRES [base] <- _ACTION [plus] @@
```



```
##################################################
# FILE: CHECKS.PAT
##################################################
@PATH _ROOT _CHECKS _NLPPP

@POST
      rfaactions(1)
      single()
@RULES
_CHECKS [base] <- _STMTS [plus] @@
```



```
##################################################
# FILE: POSTS.PAT
##################################################
@PATH _ROOT _POSTS _NLPPP

@POST
      rfaactions(1)
      single()
@RULES
_POSTS [base] <- _STMTS [plus] @@
```



```
################################################
# FILE: TMP.PAT
# SUBJ: Delete the holding rules nodes.
################################################
@NODES _ROOT

@POST
  splice(1,1)
@RULES

_xNIL <- _RULES @@
_xNIL <- _CODE @@
_xNIL <- _PRES @@
_xNIL <- _CHECKS @@
_xNIL <- _POSTS @@
```



```
################################################
# FILE: TMP1.PAT
# SUBJ: Splice the NLPPP container.
################################################
@NODES _ROOT

@POST
  splice(1,1)
@RULES

_XNIL <- _NLPPP @@
```



```
################################################
# FILE:     SELECT.PAT
################################################
@POST
      rfaselect(2)
      single()
@RULES
_SELECT [base] <- _soSELECT [opt] _NODES _eoSELECT [opt] @@
_SELECT [base] <- _soSELECT [opt] _MULTI _eoSELECT [opt] @@
_SELECT [base] <- _soSELECT [opt] _PATH _eoSELECT [opt] @@
```



```
#################################################
# FILE:     REGION.PAT
#################################################
@POST
      rfaregion(1, 2, 3, 4)
      single()
@RULES
_REGION [base] <- _PRES [opt] _CHECKS [opt] _POSTS [opt] _RULES @@
```



```
##################################################
# FILE:     REGIONS.PAT
##################################################
@POST
      rfaregions(1)
      single()
@RULES
_REGIONS [base] <- _REGION [plus] @@
```



```
##################################################
# FILE:     RECURSE.PAT
##################################################
@POST
      rfarecurse(2, 3, 5)
      single()
@RULES
_RECURSE [base] <- _soRECURSE [s] _LIT _REGIONS [opt] _eoRECURSE [s] _LIT
[opt] @@
```



```
##################################################
# FILE:     RECURSES.PAT
##################################################
@POST
      rfarecurses(1)
      single()
@RULES
_RECURSES [base] <- _RECURSE [plus] @@
```



```
################################################
# FILE:     RULESFILE.PAT
################################################
@POST
      rfarulesfile(1, 2, 3, 4)
      single()
@RULES
# ALLOWING EMPTY RULE REGION IF THERE IS A CODE REGION.
_RULESFILE [base] <- _CODE _SELECT [opt] _RECURSES [opt] _REGIONS [opt] @@

_RULESFILE [base] <- _CODE [opt] _SELECT [opt] _RECURSES [opt] _REGIONS @@
```


APPENDIX III:

A BNF GRAMMAR FOR AN INSTANTIATION OF NLP++

This appendix specifies a syntax for an embodiment of NLP++. The syntax is given in extended Backus-Naur form:

```
<_RULESFILE> ::= [ <_CODE> ] [ <_SELECT> ] [ <_RECURSES> ] [ <_REGIONS> ]
<_RECURSES>  ::= { <_RECURSE> }
<_RECURSE>   ::= <_soRECURSE> <_LIT> [ <_REGIONS> ] <_eoRECURSE> [ <_LIT> ]
<_REGIONS>   ::= { <_REGION> }
<_REGION>    ::= [ <_PRES> ] [ <_CHECKS> ] [ <_POSTS> ] [ <_RULES> ]
<_SELECT>    ::= [ <_soSELECT> ] { <_NODES> | <_MULTI> | <_PATH> } [ <_eoSELECT> ]
<_CODE>      ::= <_soCODE> { <_STMTS> } [ <_eoCODE> ]
<_PRES>      ::= <_soPRE> { <_ACTION> } [ <_eoPRE> ]
<_CHECKS>    ::= <_soCHECK> { <_STMTS> } [ <_eoCHECK> ]
<_POSTS>     ::= <_soPOST> { <_STMTS> } [ <_eoPOST> ]
<_RULES>     ::= <_soRULES> { <_RULE> } [ <_eoRULES> ]

<_RULE>      ::= <_SUGG> <_ARROW> <_PHRASE> <_ENDRULE>
<_SUGG>      ::= <_ELT>
<_PHRASE>    ::= { <_ELT> }
<_ELT>       ::= <_ELEMENT> | <_TOKEN>
<_ELEMENT>   ::= <_TOKEN> <_PAIRS>
<_TOKEN>     ::= <_NONLIT> | <_LIT> | <_NUM>
<_PAIRS>     ::= "[" { <_LIT> | <_PAIR> } "]"
<_PAIR>      ::= <_LIT> "=" <_VAL>
<_VAL>       ::= <_LIT> | <_NONLIT> | <_STR> | <_NUM> | <_LIST>
<_LIST>      ::= "(" { <_ARG> } ")"
<_ARG>       ::= <_LIT> | <_NONLIT> | <_STR> | <_NUM>

<_STMTS>     ::= { <_STMT> | <_EXPR> | <_BLOCK> }
<_BLOCK>     ::= "{" <_STMTS> "}"

<_STMT>      ::= <_IFSTMT> [ <_ELSEPART> ]
<_IFSTMT>    ::= <_IFPART> ( <_BLOCK> | <_STMT> | <_EXPR> )
<_IFPART>    ::= <_IF> ( <_EXPR> | <_NUM> | <_FLOAT> | <_STR> )
<_ELSEPART>  ::= <_ELSE> ( <_BLOCK> | <_STMT> | <_EXPR> )
```



```
<_STMT>       ::= <_WHILECOND> ( <_BLOCK> | <_STMT> | <_EXPR> ) ";"
<_WHILECOND> ::= <_WHILE> <_EXPR>

<_STMT>       ::= ( <_EXPR> | <_NUM> | <_FLOAT> | <_STR> )

<_ACTION>     ::= <_PREPAIR> <_FNCALL> ";"
<_EXPR>       ::= <_FNCALL> | <_VAR>
<_FNCALL>     ::= [ <_SCOPE> ] <_LIT> "(" [ <_FNARGLIST> ] ")"
<_SCOPE>      ::= <_LIT> ":" ":"
<_FNARGLIST> ::= <_FNARG> { "," <_FNARG> }
<_FNARG>      ::= <_EXPR> | <_NUM> | <_FLOAT> | <_STR>
<_VAR>        ::= ( "S" | "G" | "N" | "X") "(" [ <_FNARGLIST> ] ")"


<_EXPR>       ::= "(" <_FNARG> ")"
<_EXPR>       ::= <_FNARG>
                  ( "*" | "/" | "%" | <_opCONF> | "+" | "-" | "<" | ">"
                  | <_opLE> | <_opGE> | <_opEQ> | <_opNEQ> | <_opAND> | <_opOR> )
                  <_FNARG>
<_EXPR>       ::= <_VAR> "=" <_FNARG>
<_EXPR>       ::= ( <_STR> | <_EXPR> ) <_opOUT> <_FNARG>
<_EXPR>       ::= ( <_opINC> | <_opDEC> ) <_VAR>
<_EXPR>       ::= "!" <_FNARG>
<_EXPR>       ::= <_VAR> ( <_opINC> | <_opDEC> )
<_EXPR>       ::= ( "-" | "+" ) ( <_EXPR> | <_NUM> | <_FLOAT> )

<_IF>         ::= "if"
<_ELSE>       ::= "else"
<_WHILE>      ::= "while"

<_PREPAIR>    ::= "<" <_NUM> "," <_NUM> ">"

<_NODES>      ::= <_soNODES> { <_NONLIT> } [ <_eoNODES> ]
<_PATH>       ::= <_soPATH> { <_NONLIT> } [ <_eoPATH> ]
<_MULTI>      ::= <_soMULTI> { <_NONLIT> } [ <_eoMULTI> ]

<_opAND>      ::= "&" "&"
<_opOR>       ::= "|" "|"
```



```
<_opINC>      ::= "+" "+"
<_opDEC>      ::= "-" "-"
<_opEQ>       ::= "=" "="
<_opNEQ>      ::= "!" "="
<_opGE>       ::= ">" "="
<_opLE>       ::= "<" "="
<_opCONF>     ::= "%" "%"
<_opOUT>      ::= "<" "<"

<_ENDRULE>    ::= "@" "@"

<_eoPOST>     ::= "@" "@" "POST"
<_eoCHECK>    ::= "@" "@" "CHECK"
<_eoPRE>      ::= "@" "@" "PRE"
<_eoRULES>    ::= "@" "@" "RULES"
<_eoRECURSE>  ::= "@" "@" "RECURSE"
<_eoSELECT>   ::= "@" "@" "SELECT"
<_eoNODES>    ::= "@" "@" "NODES"
<_eoMULTI>    ::= "@" "@" "MULTI"
<_eoPATH>     ::= "@" "@" "PATH"
<_eoCODE>     ::= "@" "@" "CODE"

<_soPOST>     ::= "@" "POST"
<_soCHECK>    ::= "@" "CHECK"
<_soPRE>      ::= "@" "PRE"
<_soRULES>    ::= "@" "RULES"
<_soRECURSE>  ::= "@" "RECURSE"
<_soSELECT>   ::= "@" "SELECT"
<_soNODES>    ::= "@" "NODES"
<_soMULTI>    ::= "@" "MULTI"
<_soPATH>     ::= "@" "PATH"
<_soCODE>     ::= "@" "CODE"

<_NONLIT>     ::= "_" <_xALPHA>
<_LIT>        ::= <_xALPHA>
<_LIT>        ::= "\\" <_xPUNCT>
<_FLOAT>      ::= <_xNUM> "." [ <_xNUM>
<_NUM>        ::= <_xNUM>
```

```
<_ARROW>      ::= "<" "-"
```

<_STR> is a string token.

<_XALPHA> is an alphabetic token.

<_XNUM> is an integer token.

<_XPUNCT> is a punctuation character token.

APPENDIX IV:
THE CONFIDENCE OPERATOR ACCORDING TO ONE EMBODIMENT

The confidence operator combines confidence values while never exceeding 100% confidence. The confidence operator provides a way to accumulate evidence for competing hypotheses.

Mathematical functions are available or may be with the properties that infinite evidence (or some maximal quantity) equates to 100% confidence and that no evidence equates to a 0% confidence. In one embodiment:

$$P = 100 * (1 - 1/(1 + E)) \qquad (1)$$

$$E = P (100 - P) \qquad (2)$$

where P is the percentage of confidence ad E is a fabricated evidence metric that ranges from 0 to infinity.

Say, the suffice "-ence" gives a 70% confidence level that a word is a noun. Say, also, that if a word appears immediately following the word "a," there is an 85% confidence that the word is a noun. Then the accumulated confidence is 70 %% 85, some number greater than 85 and less than 100.

The probabilty of a noun, based on the suffix, is 70%. Equation (2) gives E_1, the evidence from the suffix, as 2.33. The probability of a noun, based on the preceding article "a," is 85%. Equation (2) gives E_2, the evidence from the article, as 5.66. E, the total evidence, is the sum of E_1 and E_2. Thus, E is 8.00. Equation (1) gives a probability of 88.9% with evidence E equal to 8.00.

E_1 may be based on statistical studies, a guess (educated or otherwise), gut feel, etc. The same is true of E_2. This is a standing problem in statistics. While there are many ways to generate the initial confidence numbers, typically, one starts with initial values and modifies them based on how well those values work in practice.

United States Patent & Trademark Office

Office of Initial Patent Examination -- Scanning Division

SCANNED, # 8

Application deficiencies found during scanning:

☐ Page(s) ____________ of ______________________________ were not present for scanning. (Document title)

☐ Page(s) ____________ of ______________________________ were not present for scanning. (Document title)

Pages 40 - 95 of the specification are appendix.

☒ ***Scanned copy is best available.*** Drawing figures 3 - 12 are dark